CSE 675.02: Introduction to Computer Architecture

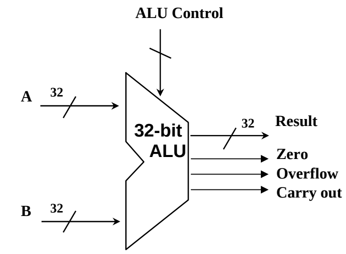
Arithmetic / Logic Unit – ALU Design

Presentation F

Slides by Gojko Babić

07/19/2005

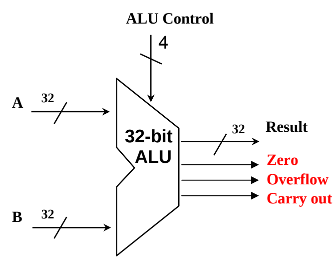
32-bit ALU



- Our ALU should be able to perform functions:
  - logical and function
  - logical or function
  - arithmetic add function
  - arithmetic subtract function
  - arithmetic slt (set-less-then) function
  - logical nor function
- ALU control lines define a function to be performed on A and B.

Functioning of 32-bit ALU

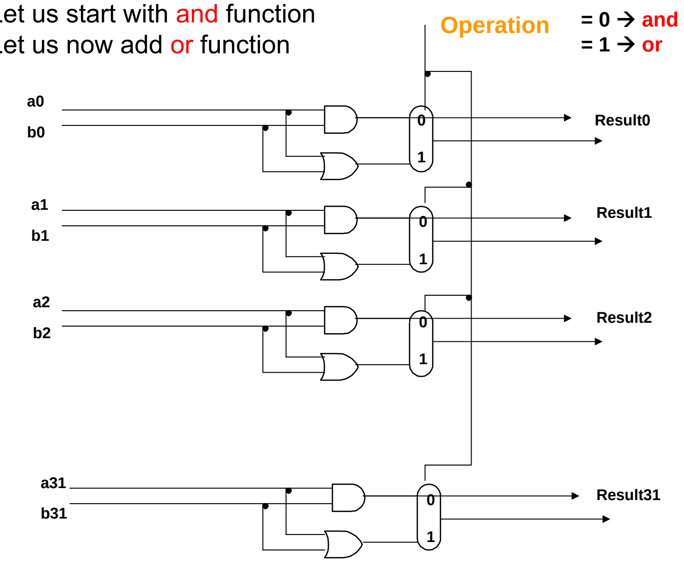
	ALU Control lines		
Function	**Ainvert**	**Binvert**	**Operation**
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00



- Result lines provide result of the chosen function applied to values of A and B
- Since this ALU operates on 32-bit operands, it is called 32-bit ALU
- Zero output indicates if all Result lines have value 0
- Overflow indicates a sign integer overflow of add and subtract functions; for unsigned integers, this overflow indicator does not provide any useful information
- Carry out indicates carry out and unsigned integer overflow

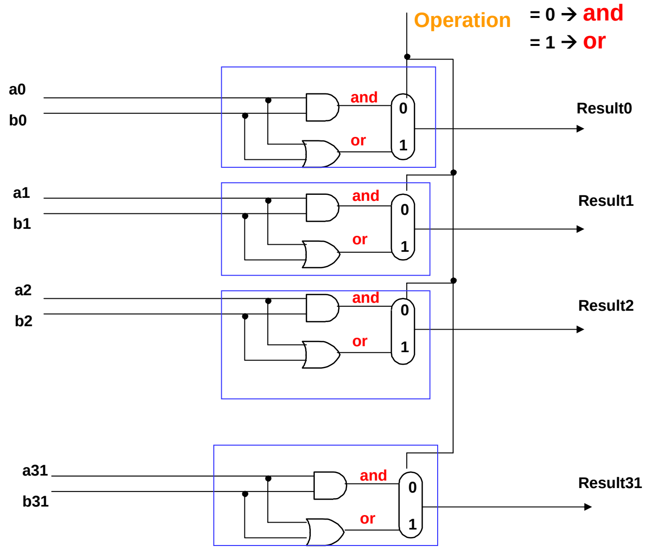
Designing 32-bit ALU: Beginning

1. Let us start with and function
2. Let us now add or function



Designing 32-bit ALU: Principles

- **Number of functions are performed internally, but only one result is chosen for the output of ALU**
- **32-bit ALU is built out of 32 identical 1-bit ALU's**



Designing Adder

- 32-bit adder is built out of 32 1-bit adders

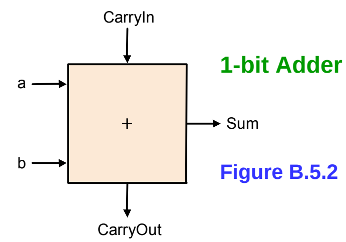


Figure B.5.2

From the truth table and after minimization, we can have this design for CarryOut

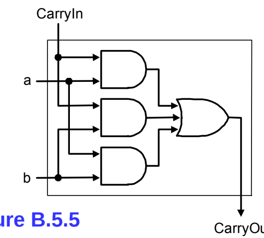
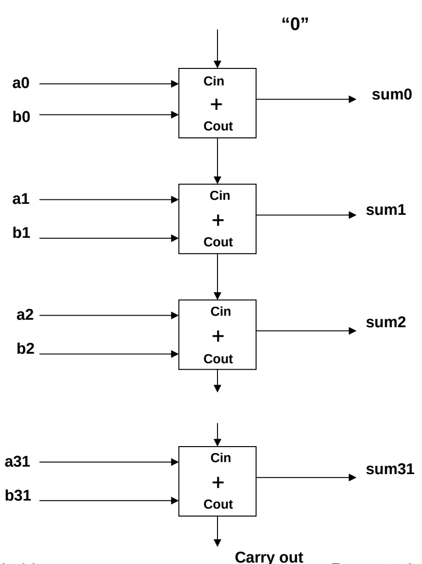


Figure B.5.5

1-bit Adder Truth Table

Input			Output	
a	b	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

32-bit Adder



This is a ripple carry adder.

The key to speeding up addition is determining carry out in the higher order bits sooner.
Result: **Carry look-ahead adder**.

32-bit ALU With 3 Functions

1-bit ALU

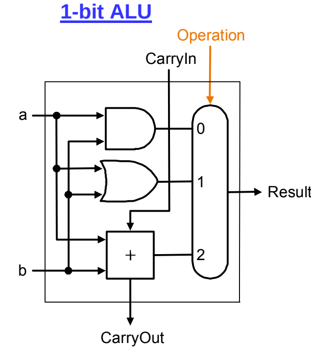
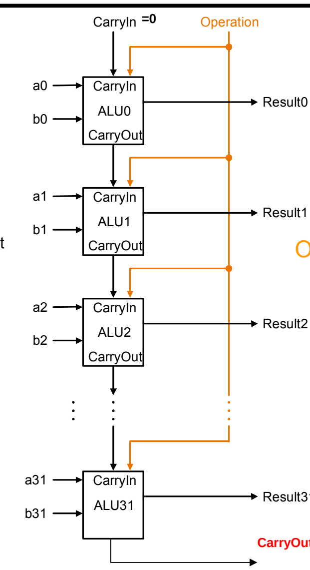


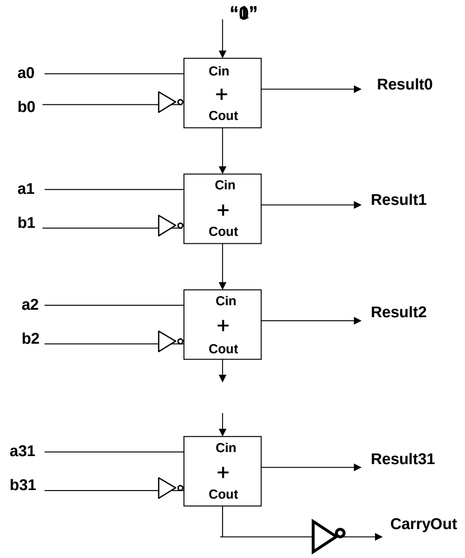
Figure B.5.6



Operation = 00 → **and**
= 01 → **or**
= 10 → **add**

Figure B.5.7
+ carry out

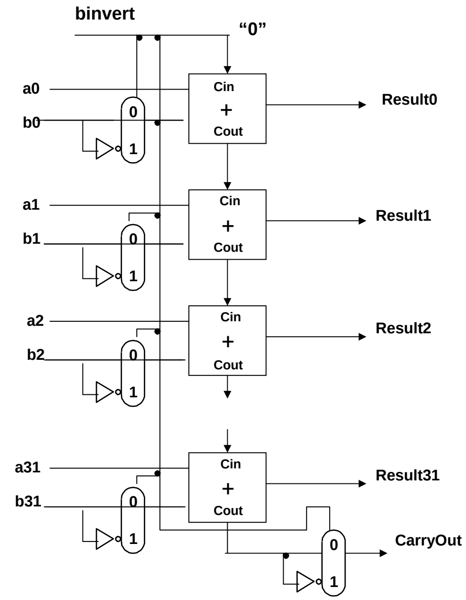
32-bit Subtractor



$$A - B = A + (-B)$$

$$= A + \overline{B} + 1$$

32-bit Adder / Subtractor



Binvert = 0 → addition
= 1 → subtraction

32-bit ALU With 4 Functions

1-bit ALU

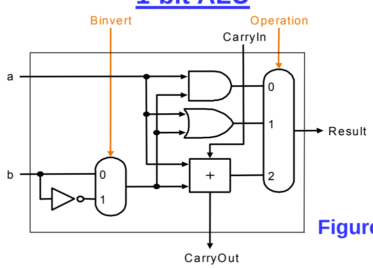
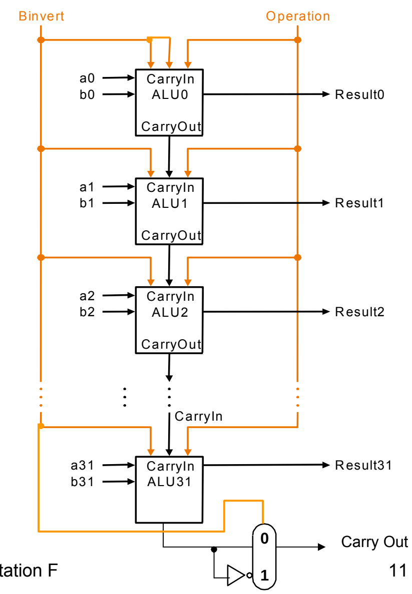


Figure B.5.8

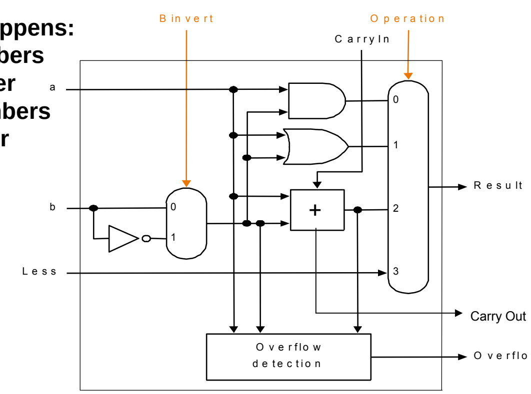
	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10



2's Complement Overflow

1-bit ALU for the most significant bit

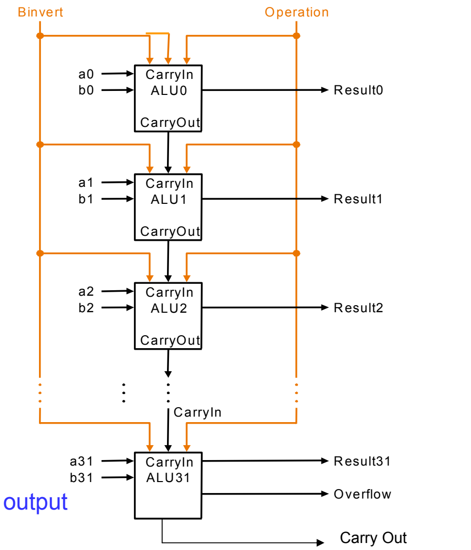
2's complement overflow happens:
- **if sum of two positive numbers results in a negative number**
- **if sum of two negative numbers results in a positive number**



Other 1-bit ALUs, i.e. non-most significant bit ALUs, are not affected.

32-bit ALU With 4 Functions and Overflow

	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10



Missing: slt & nor functions and Zero output

Add correction for CarryOut

Set Less Than (slt) Function

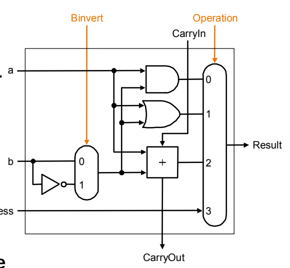
- slt function is defined as:

A slt B = $\begin{cases} 000 \ldots 001 & \text{if } A < B, \text{ i.e. if } A - B < 0 \\ 000 \ldots 000 & \text{if } A \geq B, \text{ i.e. if } A - B \geq 0 \end{cases}$

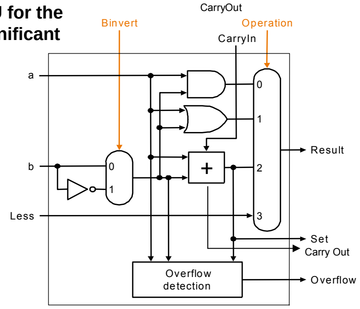
- Thus each 1-bit ALU should have an additional input (called "Less"), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit Less value should be sign of A – B

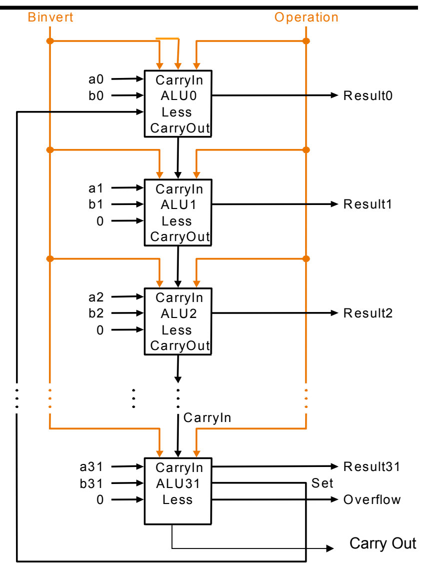
32-bit ALU With 5 Functions

1-bit ALU for non-most significant bits



1-bit ALU for the most significant bits



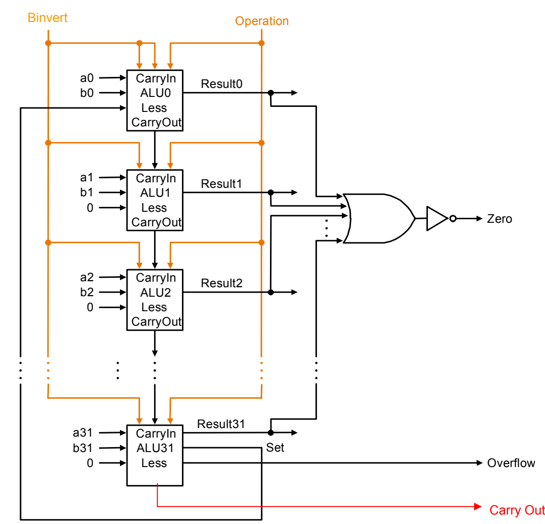


Operation = 3 and Binvert =1 for slt function

Add correction for CarryOut

32-bit ALU with 5 Functions and Zero

	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10
slt	1	11



Add correction for CarryOut

32-bit ALU with 6 Functions

A nor B = $\overline{A}$ and $\overline{B}$

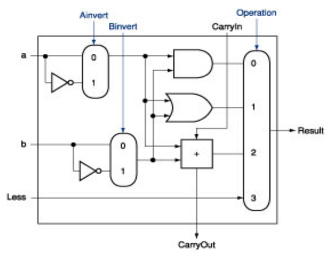
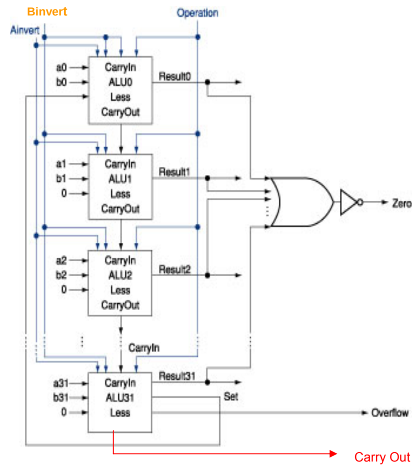


Figure B.5.10 (Top)

Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00



Add correction for CarryOut

Figure B.5.12
+ Carry Out + Binvert

32-bit ALU Elaboration

- We have now accounted for all but one of the arithmetic and logic functions for the core MIPS instruction set. 32-bit ALU with 6 functions omits support for shift instructions.
- It would be possible to widen 1-bit ALU multiplexer to include 1-bit shift left and/or 1-bit shift right.
- Hardware designers created the circuit called a barrel shifter, which can shift from 1 to 31 bits in no more time than it takes to add two 32-bit numbers. Thus, shifting is normally done outside the ALU.
- We now consider integer multiplication (but not division).

Multiplication

- Multiplication is more complicated than addition:
  – accomplished via shifting and addition
- More time and more area required
- Let's look at 3 versions based on elementary school algorithm
- Example of unsigned multiplication:

```
5-bit multiplicand     10001₂ = 17₁₀
5-bit multiplier     × 10011₂ = 19₁₀
                       10001
                      10001
                     00000
                    00000
                   10001
                  101000011₂ = 323₁₀
```

- But, this algorithm is very impractical to implement in hardware

Multiplication : Example

- The multiplication can be done with intermediate additions.
- The same example:

multiplicand	10001
multiplier	× 10011
intermediate product	0000000000
add since multiplier bit=1	10001
intermediate product	0000010001
shift multiplicand and add since multiplier bit=1	10001
intermediate product	0000110011
shift multiplicand and no addition since multiplier bit=0	
shift multiplicand and no addition since multiplier bit=0	
shift multiplicand and add multiplier since bit=1	10001
final result	0101000011

Multiplication Hardware: 1st Version

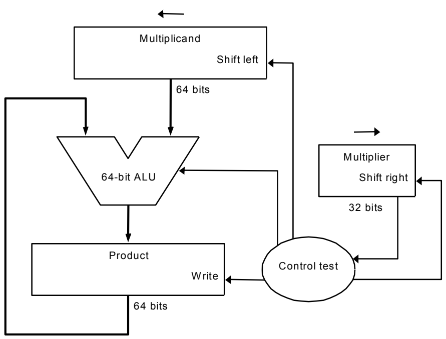


Figure 3.5

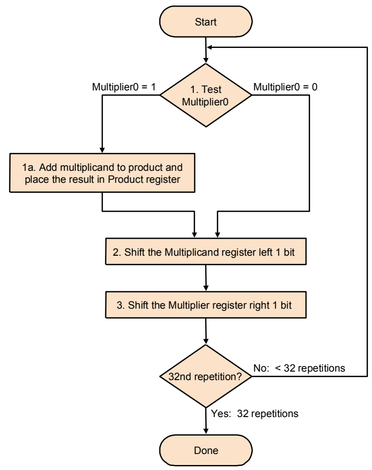
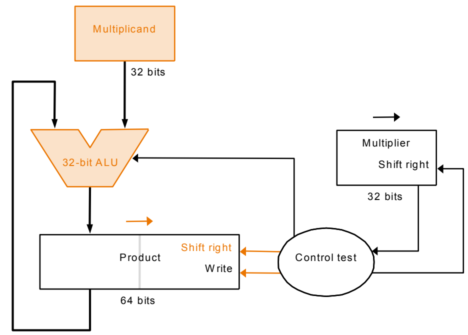
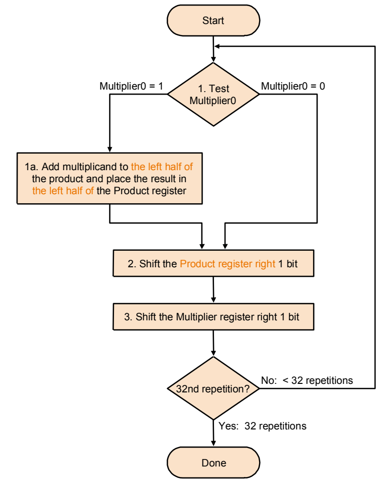


Figure 3.6

Multiplication Hardware: 2nd Version





Multiplication Hardware: 3rd Version

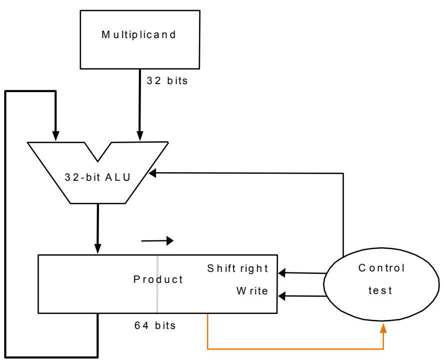
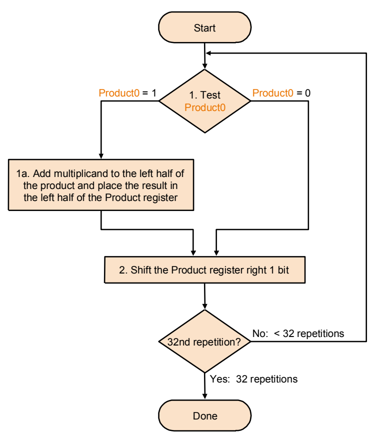


Figure 3.7



Multiplication of Signed Integers

- A simple algorithm:
  - Convert to positive integer any of operands (if needed) and remember original signs
  - Perform multiplication of unsigned numbers using the existing algorithm and hardware
  - Negate product if original signs disagree
- This algorithm is not simple to implement in hardware, since it has to:
  - account in advance about signs,
  - if needed, convert from negative to positive numbers,
  - if needed, convert back to negative integer at the end
- Fast multiplication algorithms.

Real Numbers

- *Conversion from real binary to real decimal*

  $-1101.1011_2 = -13.6875_{10}$

  since: $1101_2 = 2^3 + 2^2 + 2^0 = 13_{10}$ and

  $0.1011_2 = 2^{-1} + 2^{-3} + 2^{-4} = 0.5 + 0.125 + 0.0625 = 0.6875_{10}$
- **Conversion from real decimal to real binary:**

  $+927.45_{10} = +1110011111.01\ 1100\ 1100\ 1100\ \ldots\ldots$

927/2 = 463 + ½ ← LSB	0.45 × 2 = 0.9
463/2 = 231 + ½	0.9 × 2 = 1.8
231/2 = 155 + ½	0.8 × 2 = 1.6
155/2 = 57 + ½	0.6 × 2 = 1.2
57/2 = 28 + ½	0.2 × 2 = 0.4
28/2 = 14 + 0	0.4 × 2 = 0.8
14/2 = 7 + 0	0.8 × 2 = 1.6
7/2 = 3 + ½	0.6 × 2 = 1.2
3/2 = 1 + ½	0.2 × 2 = 0.4
1/2 = 0 + ½	0.4 × 2 = 0.8 ……

Floating Point Number Formats

- The term floating point number refers to representation of real binary numbers in computers.
- IEEE 754 standard defines standards for floating point representations
- Single precision:

31	30 … 23	22 … 0
s	E	Fraction

- Double precision:

63	62 … 52	51 … 32
s	E	Fraction

31 … 0
Fraction

Converting to Floating Point

1. Normalize binary real number i.e. put it into the normalized form:

   $(-1)^s \times 1.\text{Fraction} * 2^{Exp}$

   $-1101.1011_2 = (-1)^1 \times 1.1011011 * 2^3$

   $+1110011111.01\overline{1100} = (-1)^0 \times 1.11001111101\overline{1100} * 2^9$

2. Load fields of single or double precision format with values from normalized form, but with the adjustment for E field.

   $E = Exp + 127_{10} = Exp + 01111111_2$ for single precision
   $E = Exp + 1023_{10} = Exp + 01111111111_2$ for double precision

- E is called a biased exponent.

Floating Point: Example 1

- Find single and double precision of -13.6875_{10}

  Normalized form: $(-1)^1 \times 1.1011011 \times 2^3$

  – single precision:

  $E = 11_2 + 01111111_2 = 10000010_2$

  |1|10000010|10110110000000000000000|

  – double precision

  $E = 11_2 + 01111111111_2 = 10000000010_2$

  |1|10000000010|10110110000000000000|

  |00000000000000000000000000000000|

Floating Point: Example 2

- Find single and double precision of $+927.45_{10}$

  Normalized form: $(-1)^0 \times 1.11001111101\overline{1100} * 2^9$

  – single precision

  $E = 1001_2 + 01111111_2 = 10001000_2$

  <u>|0|10001000|11001111101110011001100|</u>1100...

  truncation <u>|0|10001000|11001111101110011001100|</u>

  rounding <u>|0|10001000|11001111101110011001101|</u>

  – double precision

  $E = 1001_2 + 01111111111_2 = 10000001000$

  <u>|0|10000001000|11001111101110011001|</u>

  <u>|10011001100110011001100110011001|</u>1001100…

truncation <u>|10011001100110011001100110011001|</u>

rounding <u>|10011001100110011001100110011010|</u>

Converting to Floating Point: Conclusion

- Rules for biased exponents in single precision apply only for real exponents in the range [-126,127], thus we can have biased exponents only in the range [1,254].
- The number 0.0 is represented as S=0, E=0 and Fraction=0. The infinite number is represented with E=255. There are some additional rules that are outside our scope.
- Find the largest (non-infinite) real binary number (by magnitude) which can be represented in a single precision.
  – Floating point overflow
- Find the smallest (non-zero) real binary number (by magnitude) which can be represented in a single precision.
  – Floating point underflow

Floating Point Addition

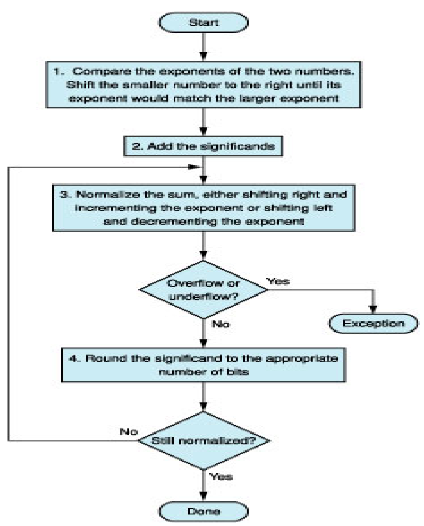


Figure 3.16

Arithmetic Unit for Floating Point Addition

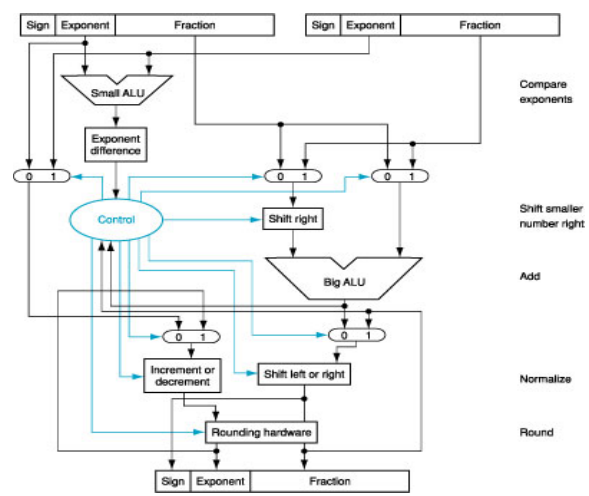


Figure 3.17

32-bit ALU with 6 Functions